ParHyFlex: A Framework for Parallel Hyper-heuristics[1]

Willem Van Onsem Bart Demoen

KU Leuven
Department of Computer Science

Abstract

A framework called *ParHyFlex* and its underlying principle are presented. *ParHyFlex* is based on the sequential *HyFlex* framework and supports the implementation of different hyper-heuristics in a parallel setting which the programmer does not need to be aware of. Its most novel feature is the way the search space of a process is influenced by experience learned by other processes. *ParHyFlex* was tested on the Maximum Satisfiability Problem where it gives good speedups. While *ParHyFlex* cannot compete with tailor-made solvers for most problems, it offers a framework for specifying new hyper-heuristics as well as a parallel environment for solving new problems.

1 Introduction

Informally, an *optimization problem* consists in finding a configuration satisfying a constraint, and such that an *objective function* has a minimal value for that configuration. More formally, for the purpose of this work, we define:

Definition 1 (Optimization problem)**.** An *optimization problem* P consists of a tuple $\langle \mathcal{C}, h, f \rangle$ with $\mathcal{C}$ a set of *configurations*, $h : \mathcal{C} \to \{\text{true}, \text{false}\}$ a *hard constraint* and $f : \mathcal{C} \to \mathbb{R}$ an *objective function*. The configurations satisfying the constraint, that is the set $h^{-1}(\{true\})$, are named *solutions*, denoted by $\mathcal{S}$. The goal is to find a solution with a minimal value for the objective function f. The result of an objective function applied to a solution s is sometimes called the *fitness-value* of s.

Often, it is too hard to find a minimal solution, e.g. when the problem is NP-hard, so in practice, one relies on approximate methods for finding a solution that is *good enough*. Such approximate methods can use heuristics like *Simulated Annealing (SA)*[6], *Genetic Algorithms (GA)*[5] and *Tabu Search (TS)*[4].

The next step in constructing solvers for optimization problems is the use of *hyper-heuristics*. Heuristics require the design of *transition functions*: functions that transform one solution into another one. Developing and combining such functions requires skill and experience. A hyper-heuristic aims to solve this problem by combining basic transition functions into more complex ones, hereby eliminating the need for lots of expert knowledge. Moreover, a hyper-heuristic is problem independent and because it learns on the fly which combinations of basic transition functions are successful in finding a good solution, it has a high chance to work well on many problems. On the other hand, this learning ability requires additional computational effort [1, 2], and can slow down the search.

As an orthogonal way to improve a solver, one can parallelize it: the goal can be to find an acceptable solution faster, find better solutions within the same (elapsed) time or to make the system more robust. Parallelization can be done at the level of an exact tailor-made algorithm, or at the level of the hyper-heuristics framework[2]: the latter is the subject of our research. The sequential *HyFlex* framework

[1] Most of this research was done in fulfillment of the requirements for the degree of Master in Computer Science by the first author.

[2] and anything in between

[3] was the inspiration to build - from scratch - our own *ParHyFlex* framework, a parallel version of *HyFlex*. As a test case, we have used it to implement solvers for four problems. In Section 2, the *ParHyFlex* framework is introduced, its components, the concepts that are used in it, and some of its inner workings. The novel concept of *enforceable constraint* and how it is used to guide the *search space* of each process is particularly important. Section 3 describes related work and how *ParHyFlex* differs. Section 4 discusses the benchmark results on the MAX-SAT problem. Section 5 concludes and indicates future work.

2 The *ParHyFlex* framework

We start with a brief introduction of the *HyFlex* system, as it was the starting point of our own work.

2.1 *HyFlex*

In 2009 *Burke et al.*[3] published a framework implemented in Java supporting the implementation of hyper-heuristics. One implements a problem by providing an implementation of the solution representation, a set of transition functions, and an objective function: the latter two are called by a hyper-heuristic. In this way, the framework solves the problem without needing any knowledge about the problem or functions themselves. Therefore the hyper-heuristic can be used to solve a large range of optimization problems.

2.2 Overview of *ParHyFlex*

ParHyFlex[3] is a variant of the *HyFlex* framework intended to make it easy to run hyper-heuristics in parallel. Using the framework, one can implement a hyper-heuristic without any knowledge about the algorithm running on parallel systems. The framework was implemented from scratch in Java. Communication is handled using the popular Message Passing Interface (MPI) protocol. The protocol enables to process data on different machines, on the cores of one machine, or a hybrid topology.

The implementation uses a peer-to-peer philosophy where each process runs its own (potentially different) hyper-heuristic. The framework provides a communication mechanism enabling the hyper-heuristics running in different processes to exchange data and cooperate with each other. As it is often profitable to make processes explore disjunct subsets of the solution space, *ParHyFlex* provides a mechanism to define such subsets, and to confine a process to them. This mechanism is based on the novel notion of *enforceable constraints* explained in Definition 2.

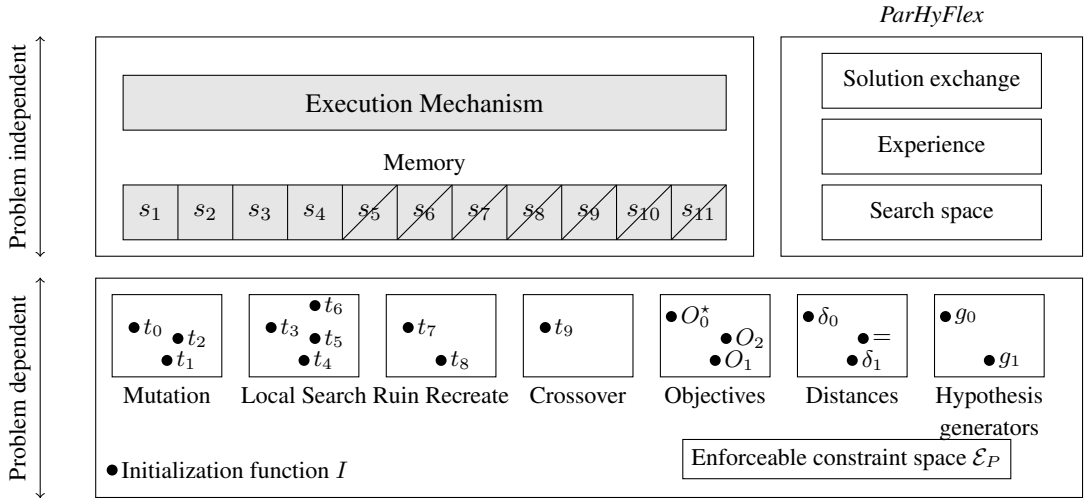

Figure 1: Structure of the *ParHyFlex* framework.

[3]The source code is available at `http://goo.gl/AyvHA`.

Figure 1 shows the structure of the framework. The lower components specify a particular problem and must be provided by the programmer interested in solving that problem: its components are explained in Section 2.7. The upper components solve a problem by using the lower components, but without any knowledge of the problem itself. The main component is the *execution mechanism* which is an implemented hyper-heuristic. Each process runs its own framework and thus its own execution mechanism. The framework provides components for *solution exchange* between processes (see Section 2.3), the generation (Section 2.4) and maintenance of an *experience set* (Section 2.6), and the *exchange of experience* for controlling the local *search space* (Section 2.5).

The explanation of these components is based on the following definitions, in which the problem $P = \langle \mathcal{C}, h, f \rangle$ is implicit.

Definition 2 (Enforceable constraint)**.** An *enforceable constraint* e is a tuple $\langle c, e^+, e^- \rangle$ with $c : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ a constraint[4], $e^+ : \mathcal{S} \rightarrow \mathcal{S}$ a function mapping each solution to a solution satisfying the constraint c, and $e^- : \mathcal{S} \rightarrow \mathcal{S}$ a function mapping each solution to a solution that does not satisfy the constraint c. We say that a solution *satisfies* an enforceable constraint $\langle c, e^+, e^- \rangle$ when it satisfies c. The set of all enforceable constraints for a given problem P is denoted by $\mathcal{E}_P$.

Within *ParHyFlex*, enforceable constraints are crucial to force two processes to explore different subsets of the solution space: by applying e^+ and e^- to a solution, one obtains two solutions that differ in the fact that they do or do not satisfy c. Augmenting the problem locally with opposite enforceable constraints thus divides the search space in different subsets. Details regarding this process are explained in Section 2.5.

In order to generate such enforceable constraints, we need a function that knows about the specific problem.

Definition 3 (Hypothesis generator)**.** A *hypothesis generator* g_P is a problem dependent partial function $g_P : \mathcal{P}(\mathcal{S}) \rightarrow \mathcal{E}_P$ that generates an enforceable constraint from one or more solutions.

The idea is that from a set of solutions, possibly obtained using a particular meta-heuristic, a common characteristic of these solutions is captured by the constraint c in the generated enforceable constraint $\langle c, e^+, e^- \rangle$. e^+ and e^- can now be used to divide the solution set.

With *experience set*, we mean a set of enforceable constraints as maintained by *ParHyFlex* in a particular way: this is explained in Section 2.4.

We also define our notion of *search space*: it is based on enforceable constraints and meant to give a means to operationally delimit the neighborhood in which a process can search for solutions.

Definition 4 (Search space)**.** A *search space* $\mathcal{A}$ is a subset of the set of solutions $\mathcal{A} \subseteq \mathcal{S}$. A search space is represented by a tuple $\langle A^+, A^- \rangle$ where A^+ and A^- are sets of enforceable constraints, named respectively the positive and the negative set. $\mathcal{A}$ contains a solution s if and only if s satisfies at least one enforceable constraint in A^+, and s does not satisfy any of the enforceable constraints in A^-.

Finally, we need the notion of *correcting* a solution, in case it does not belong to the intended search space:

Definition 5 (Corrected solution)**.** A solution s' is *corrected from* $s \notin \mathcal{A}$ *with respect to* $\mathcal{A}$ when s' is the result of the application of the positive transition function e^+ of any enforceable constraint $e \in A^+$, followed by the application of the negative transition function e^- of every enforceable constraint $e \in A^-$.

2.3 Exchanging solutions

The so-called *island model* has been successfully used in the implementation of parallel genetic algorithms [14]: possibly different algorithms work on isolated populations of solutions and occasionally exchange solutions are exchanged among several populations. *ParHyFlex* adheres to the same island

[4] not to be confused with the hard constraint of the problem

model: a hyper-heuristic chooses among a number of *exchange strategies* provided by the framework. An exchange strategy specifies when a solution is exchanged together with the set of processes receiving that solution. Solutions are exchanged through unreliable asynchronous communication: for correctness, there is no need that solutions arrive at all, or in order.

2.4 Generating experience

Modern SAT solvers try to speed up the search for a solution by learning new clauses on the fly (see for instance [10]). Such clauses can be considered as *search experience*. In the parallel context, sharing of such experience usually reduces the search in several parts of the search tree and thus leads to an overall speedup.

A similar principle is implemented in *ParHyFlex* using enforceable constraints. Enforceable constraints are generated by a hypothesis generator, based on previously obtained solutions. They are then used to restrict the set of considered solutions. The hypothesis generator needs knowledge of the problem, and is therefore provided by the user of *ParHyFlex*, not by the framework itself. Since enforceable constraints are generated from a limited set of solutions, there is no guarantee that the optimal solution satisfies the generated enforceable constraints, i.e. a hypothesis generator merely tries to generate enforceable constraints that are probably true.

The enforceable constraints generated by a hypothesis generator make up the experience set. The experience set maintains a fixed number of items and occasionally eliminates items. This procedure is discussed further in Section 2.6.

2.5 Controlling the search space by exchanging experience

Once a set of enforceable constraints is generated and maintained in the experience set, it is used to restrict the part of the solution space that is further explored. The positive set of the search space local to a process takes items from the process's experience set. Complementary to that, the negative set of a search space of one process can take items from the experience set of a different process. This exchange of experience happens occasionally. In this way, different processes are prevented from exploring the same search spaces, reaching similar solutions, and getting stuck in the same local optimum. Moreover, in this way, the search space of every process can evolve over time.

The process of correcting a solution with respect to $\mathcal{A}$ is meant to result in a solution belonging to $\mathcal{A}$. However, the application of an e^- from one enforceable constraint can revert the effects of another e^+: this is not handled by the *ParHyFlex* framework. Therefore, a search space is only enforced in a fuzzy way: occasionally corrected solutions will not be part of the intended search space.

The search space concept aims at speeding up the convergence towards the optimal solution, but a search space can become too restricted. Since one wants a hyper-heuristic to find the optimal solution eventually, the search space is occasionally regenerated.

2.6 Maintaining experience

Generated enforceable constraints are not guaranteed to be satisfied by the optimal solution. Therefore the *ParHyFlex* framework monitors continuously whether new solutions satisfy the current experience: depending on the fitness-value of a new solution, the system evaluates the enforceable constraints. Occasionally the system removes a number of enforceable constraints that are not satisfied by good solutions.

Even when newly generated solutions satisfying a particular enforceable constraint yield higher fitness-values, this does not guarantee that this enforceable constraint is true for the optimal solution(s). Indeed, a large set of suboptimal solutions can satisfy the enforceable constraint while better solutions do not. Since the enforceable constraints in the experience set eventually form the search space, such enforceable constraints can prevent further improvement. Therefore the framework can also eliminate enforceable constraints that are considered to perform well. The system uses a probabilistic mechanism where the probability of elimination is proportional to the performance of the constraint. Eventually, every enforceable constraint is eliminated.

2.7 Implementing a problem

The lower components in Figure 1 show the different modules a programmer should implement when specifying a problem. Naturally, one needs to define a set of transition function. These functions are divided into four categories: *mutation*, *local search*, *ruin recreate* and *crossover*. This distinction originates from the *HyFlex* framework. One does not need to implement transition functions of all kinds: the categories only aim to help the hyper-heuristic in deciding which function should be applied.

Furthermore, the framework supports *multiobjectivisation*. Multiobjectivisation is a process where one objective is split into more objectives. Since a significant amount of computational resources is spent on escaping from a local optimum, one could use another objective in the hope the algorithm didn't reach a local optimum according to the second objective. A problem however with this approach is that the objectives can cancel each other out. Therefore, most algorithms use a weighted sum of the different objectives. Estimating these weights however, is a nontrivial task. The *ParHyFlex* facilitates multiobjectivisation: the user must define the main objective $O_0^\star$ together with zero or more other objectives O_i, but it is up to the hyper-heuristic to deal with the different objectives.

2.8 Execution pipeline

When the framework is started, the different processes acquire the problem, i.e. the tuple $\langle \mathcal{C}, h, f \rangle$. The memory of each process is initialized with some local solutions, using the initialization function I. A process also has separate memory for the *foreign* solutions, i.e. the ones received from other processes: in the figure, they are shown as a barred s_i in memory. The generation of the initial solutions has a random component, so different processes start in a different initial state. Moreover, the hyper-heuristic can in principle differ from process to process.

After this initialization, control is passed to the hyper-heuristic. The hyper-heuristic decides on applying transition functions and objective functions on the solutions. The hyper-heuristic can also restart from the (or a new) initial solution, for instance when the search gets stuck and the algorithm does not see an opportunity for fast improvement. Each time a new solution is generated the solution is corrected (with respect to the local search space) and the hypothesis generators are called. The resulting enforceable constraints are used for maintaining the experience set.

Each process checks asynchronously for messages from other processes: received solutions are corrected and stored in memory. The hyper-heuristic can use these solutions for instance as a parent for the available *crossover* heuristics. When the systems generate a new search space, enforceable constraints are exchanged and are used in the negative set of the new search space. Furthermore a hyper-heuristic can distribute data among processes as well: for instance records describing the performance of the different heuristics. After dealing with any messages, the hyper-heuristic regains control, and restarts applying transition functions.

Eventually a stop criterion is reached. This could be a global criterion (e.g. wall clock time) which stops all processes, or it could be a local criterion (e.g. the fitness-value is good enough in one process). This results in outputting the best solution so far, and stopping all processes.

3 Related work

The research in parallel hyper-heuristics is currently quite sparse. One work is relatively close to ours: *Leon, Miranda and Segura*[7] implemented a parallel hyper-heuristic using a *multi-objective self-adaptive Island-based model*.

The framework combines three concepts: the *master-slave* paradigm, the island-based model (discussed in Section 2.3) and multiobjectivisation (as described in Section 2.7). The framework considers two types of processes: *worker* and the *master island*. The workers perform genetic algorithms. Each time a local stop criterion is met, the genetic algorithm sends its results to the master island. The master island then assigns a new task to the worker. A task contains a description of the genetic operator the worker should use together with the weights attached to the different objectives. The workers exchange solutions through the master island. The topology determining the set of workers that exchange solutions is modified in a dynamic way.

In [7], the framework is tested with the *Antenna Positioning Problem (APP)*[11] and the *Frequency Assignment Problem (FAP)*[12]. The researchers report a *speedup* over a sequential version. As ex-

pected from a hyper-heuristics based framework, the results are not on par with tailor-based sequential approaches.

The main differences with the system of [7] and our work is that *ParHyFlex* uses a peer-to-peer philosophy where the processes communicate in asynchronous way without a master process. Furthermore, *ParHyFlex* explicitly restricts the search space local to a process in a cooperative way, which is, to the best of our knowledge, the first parallel hyper-heuristic that aims to do this. Finally in *ParHyFlex* one can implement any kind of hyper-heuristic instead of using an evolutionary algorithm.

4 Benchmark results

In order to test the framework, we have implemented a hyper-heuristic based on the *AdapHH*[8] algorithm of Mustafa Mısır. The algorithm was tested on the following problems: *Maximum Satisfiability*, *Frequency Assignment*, *Antenna Positioning* and *Circle Positioning*. We report here only on the benchmark results for Max-Sat. The tests were performed on machines with an *Intel i5 2400* processor[5] and 3.7 GiB memory running the *Ubuntu 12.04* operating system. The network uses *Gigabit Ethernet* and is organized with a centralized switch. The timings are shown in seconds.

Evaluating the performance of a parallel hyper-heuristic is cumbersome. The underlying heuristics are non-deterministic, and a seed-based implementation is almost impossible. Different runs give different results and the objective function varies in time. Therefore there is no strict definition of speedup in this context. Finally since hyper-heuristics like *AdapHH* come with a number of parameters, one needs to run experiments over a large number of settings.

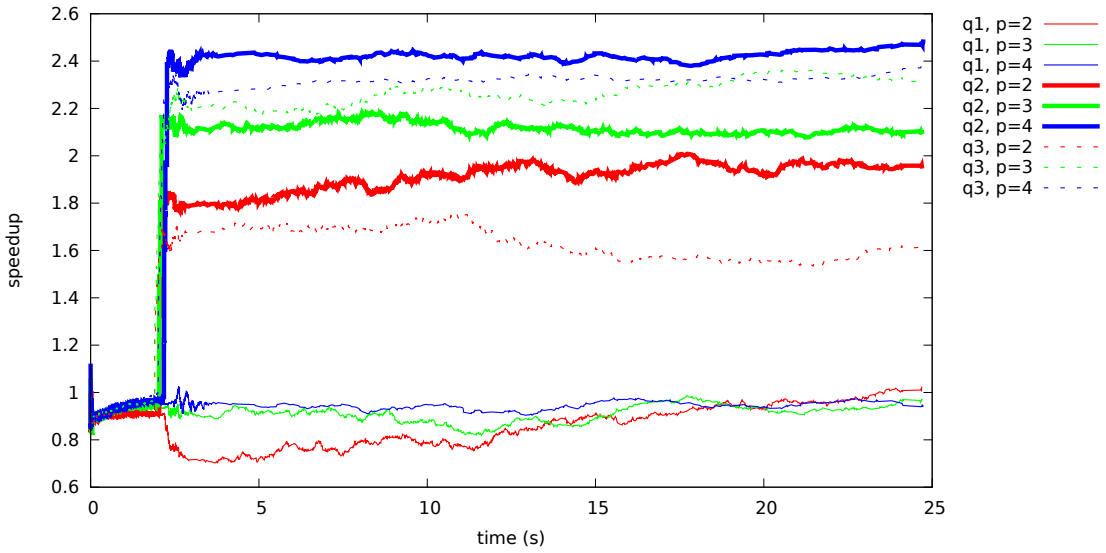

Figure 2: The evolution of the speedup with a different number of processors.

Figure 2 shows the speedup obtained by 476 runs per processor configuration solving 25 instances of the Maximum Satisfiability problem with $10'000$ variables and $42'600$ clauses, and for 2 up to 4 processes. We calculate the speedup by measuring the time at which a fraction of the runs on a certain processor configuration achieves a certain quality (the number of succeeding constraints) and compare that time with the time the same percentage of sequential runs need to achieve this level. More formally the speedup at time t with p processes for the q-th quartile is calculated by:

$$\text{speedup}\,(t,p,q) = \frac{\min g_{1,q}^{-1}\left(g_{p,q}\left(t\right)\right)}{t} \tag{1}$$

To calculate $g_{p,q}\left(t\right)$, we determine for each of the runs with p processes the best fitness-value at time t: $g_{p,q}\left(t\right)$ is then the q-th quartile of these values.

[5] 4×3.10 GHz , 6 MiB cache.

As one can see the speedup of the first quartile is not significant and sometimes values below 1 are registered. This is due to the fact that the first quartile mainly covers runs on easy problems. Therefore it is difficult to achieve speedup since communicating data is probably equally profitable as continuing search.

In most cases the third quartile has a lower speedup than the second one. This can be explained since we expect exchanging solutions and redefining the search space is more useful when at least one or more cores have generated good solutions, something which is less likely in the context of a hard problem. In the case of premature exchange, this could result in the fact that processors are put on the wrong track. Deciding at which moment exchanging solutions becomes profitable remains however an open problem.

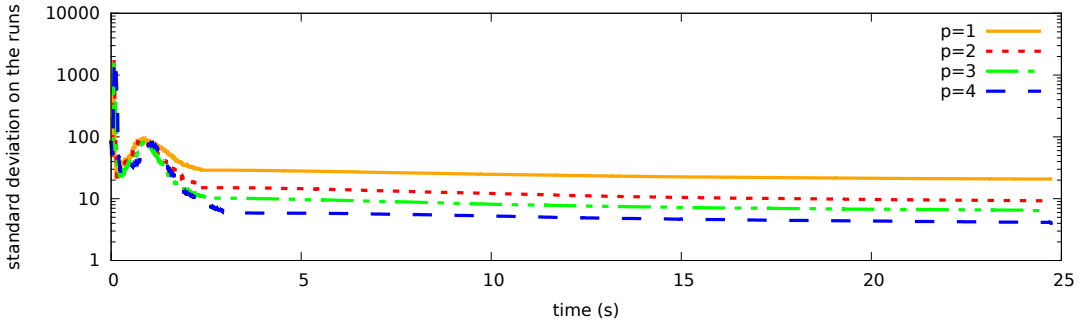

Figure 3: The evolution of the standard deviation on the fitness-value for different numbers of processes.

Figure 3 shows the standard deviation of the fitness-values of the runs per processor configuration. As one can see, the standard deviation decreases over time. This is reasonable since with a large number of parallel searches, it is less likely all processes getting stuck and thus diverging from the average.

These figures show that - at least for a small number of processors - the cooperation between the processes as provided by the *ParHyFlex* framework speeds up finding better solutions and with higher reliability.

5 Conclusions and future Work

A parallel implementation of the *AdapHH*[8] algorithm was presented, together with benchmark results for the MAX-SAT problem. *ParHyFlex* adopts a novel way to encapsulate experience from a single process in the form of enforceable constraints, and uses the exchange of experience for influencing the other processes' search space, so that there is less chance that they get stuck in similar regions of the configuration space. In our experience, *ParHyFlex* allows to specify new hyper-heuristics relatively easily, as well as new problems. We are currently experimenting with the Frequency Assignment, Circle Positioning and Antenna Positioning problems that were used in [7], and we need to extend our experience to a larger number of processes. We would also like to make a formal model of the parallel hyper-heuristics framework, and in particular study the relation with speedup prediction theory as for instance in [13].

The performance of a hyper-heuristic depends on the underlying transition functions (see [9]). Therefore it is hard to prove that the obtained results are not just a lucky coincidence between the parallel algorithm and the transition functions. In order to understand this better, we intend to implement a compiler that from a logical specification of the problem $\langle \mathcal{C}, h, f \rangle$ generates transition functions without any interaction of the programmer. In this way, we hope to obtain more objective results proving the reported speedup for a variety of problems.

Many decisions within our framework are under the control of parameters that are often evolving during the solving process. More effort is needed to understand the role of these parameters.

Finally, we intend to work on automatic specialization of our framework to specific problems, and on applying the framework to new problems e.g. for the generation of secure, low-energy, small area integrated circuits.

References

[1] E. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, J.A. Vázquez-Rodríguez, and M. Gendreau. Iterated local search vs. hyper-heuristics: Towards general-purpose search algorithms. In *IEEE Congress on Evolutionary Computation (CEC), 2010*, pages 1–8, 2010.

[2] Edmund Burke, Emma Hart, Graham Kendall, JimNewall, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology, 2003.

[3] Edmund K. Burke, T. Curtois, M. Hyde, G. Kendall, G. Ochoa, S. Petrovic, and J. A. Vazquez-Rodriguez. HyFlex: A Flexible Framework for the Design and Analysis of Hyper-heuristics. In *Multidisciplinary International Scheduling Conference (MISTA 2009), Dublin, Ireland*, pages 790–797, Dublin, Ireland, 2009.

[4] Fred Glover. Tabu Search - Part I. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, New York, 1989.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[7] Coromoto Leon, Gara Miranda, and Carlos Segura. Parallel hyperheuristic: a self-adaptive island-based model for multi-objective optimization. In Conor Ryan and Maarten Keijzer, editors, *GECCO*, pages 757–758. ACM, 2008.

[8] Mustafa Mısır, Patrick De Causmaecker, Greet Vanden Berghe, and Katja Verbeeck. An adaptive hyper-heuristic for CHeSC 2011. In *CHeSC 2011*, 2011.

[9] Mustafa Misir, Katja Verbeeck, Patrick De Causmaecker, and Greet Vanden Berghe. The effect of the set of low-level heuristics on the performance of selection hyper-heuristics. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *PPSN (2)*, volume 7492 of *Lecture Notes in Computer Science*, pages 408–417. Springer, 2012.

[10] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, February 2011.

[11] Carlos Segura, Yanira Gonzlez, Gara Miranda, and Coromoto Len. Parallel hyperheuristics for the antenna positioning problem. In AndrePonce Leon F. de Carvalho, Sara Rodrguez-Gonzlez, JuanF. Paz Santana, and JuanM.Corchado Rodrguez, editors, *Distributed Computing and Artificial Intelligence*, volume 79 of *Advances in Intelligent and Soft Computing*, pages 471–479. Springer Berlin Heidelberg, 2010.

[12] Carlos Segura, Eduardo Segredo, and Coromoto León. Scalability and robustness of parallel hyperheuristics applied to a multiobjectivised frequency assignment problem. *Soft Comput.*, 17(6):1077–1093, 2013.

[13] Ron Shonkwiler and Erik S. Van Vleck. Parallel Speed-Up of Monte Carlo Methods for Global Optimization. *J. Complexity*, 10(1):64–95, 1994.

[14] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–47, 1998.